Homework 7 Selected Solutions
APPM/MATH 4650 Fall '20 Numerical Analysis

Due date: Saturday, October 24, before midnight, via Gradescope. **Instructor**: Prof. Becker
Theme: Gaussian quadrature and other numerical integration

solutions version 10/26/2020

Problem 1: Gauss-Laguerre quadrature We'll estimate the integral of the Runge function over the whole real line

$$I = \int_{-\infty}^{\infty} \frac{1}{1+x^2}\,dx = 2\int_0^{\infty} \frac{1}{1+x^2}\,dx$$

using Gauss-Laguerre quadrature. Note that the Runge function is the probability density function (pdf) of the Cauchy distribution (up to a normalization constant), so the analytic solution is known; in fact, the antiderivative of $\frac{1}{1+x^2}$ is $\tan^{-1}(x)$, so we can use this to check our answer. Background on Gauss-Laguerre is at the end of this document.

a) Using a software library such as `lagpts.m` (Matlab) or `scipy.special.roots_laguerre` (Python) [see the background section at the end of this document], for various values of n, compute the nodes and weights for Gauss-Laguerre quadrature and use these to estimate I[1]. How small can you make the error (and for which value of n is this)? Can you make the error as close to machine precision as you like?

Solution:

We'll work with $\int_0^\infty \frac{1}{1+x^2}\,dx$ for simplicity, but recall we should really multiply this by 2 to get the full integral over $(-\infty,\infty)$. Students can do either way.

Let $f(x) = \frac{1}{1+x^2}$. We will write $\varphi(x) = e^x f(x)$, so therefore

$$\int_0^\infty f(x)\,dx = \int_0^\infty \varphi(x)e^{-x}\,dx$$

and we can apply Gauss-Laugerre quadrature to φ.

Note: if you forgot to multiply by the e^x term and integrated without that, you actually get a reasonable error (since e^x isn't too large for small x, and most of the interesting stuff happens near $x \approx 0$), but you'll find that you can never get the error as low as you like.

As we increase n, the error starts to go down and everything looks good... for a while. See Fig. 1 left. Then at $n = 186$ we get an `Inf` value, and for $n \geq 187$ we get `NaN`. What happened?

The computation of the weights and node locations are done correctly, but for $n = 186$, the largest node is 712.6441. The issue is that $\varphi(712)$ gives us overflow. We can compute $\varphi(709) = 1.6349e+302$, but not $\varphi(710)$ or any larger input. `realmax` is $1.7977e+308$ and thus $\varphi(712.6441)$ gives overflow; this number is simply so large that we cannot store it on the computer.

So the answer to the question is that the smallest error is for $n = 185$ and here the error (in absolute value) is $\boxed{.0014}$, or $\boxed{.0028}$ is you were really doing the $(-\infty,\infty)$ interval. We $\boxed{\text{cannot make this error as small as we like}}$ since increasing n leads to numerical issues as we just discussed.

[1] *Hint*: you might want to check that your code is correct by integrating an easier function. I suggest integrating $\int_0^\infty \frac{1}{\sqrt{2\pi}} e^{-x^2/2}\,dx = 1/2$. You should get excellent results with fewer than 100 nodes.

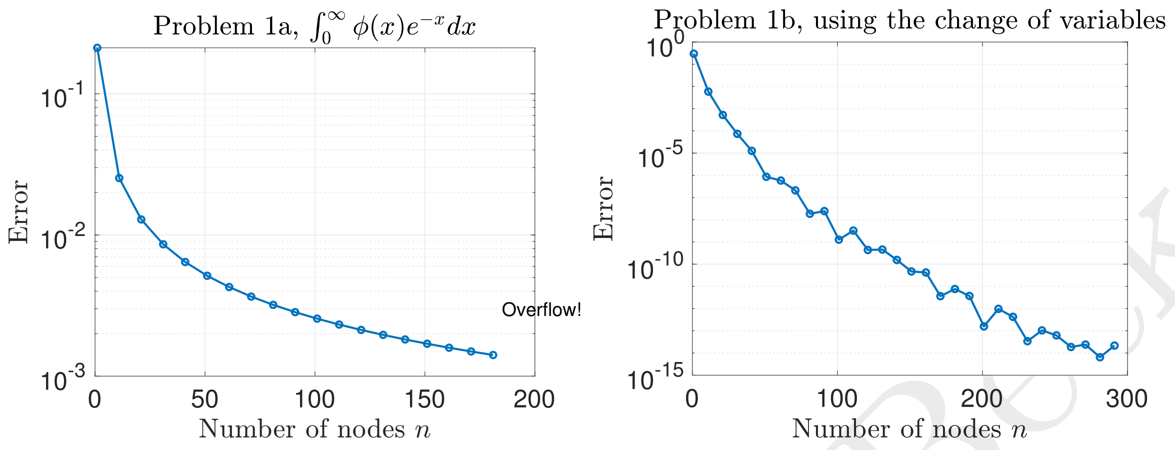


Figure 1: Left: problem 1a, where Gauss-Laguerre works until we get overflow at $n = 186$. Right: problem 1b, using Gauss-Lagerre with the change-of-variables to avoid overflow, and with the side-benefit of it working very well

Basic code in Matlab to compute the integral is shown below (the code used for plotting and trying different values of n is not shown, as this is straightforward)

```matlab
f = @(x) 1./(1+x.^2); % integrand
F = @(x) atan(x);     % antiderivative
a = 0; b = Inf;       % limits of integration
I = F(b) - F(a);      % pi/2
websave('lagpts.m','https://github.com/chebfun/chebfun/raw/master/lagpts.m')
websave('gammaratio.m','https://github.com/chebfun/chebfun/raw/master/
    gammaratio.m');
phi = @(x) exp(x).*f(x)
n       = 185;                  % or whichever value you've chosen
[nodes,weights] = lagpts(n);
Q       = dot( weights, phi(nodes) );   % Gauss-Laguerre quadrature
```

The equivalent Python version is below:

```python
import numpy as np
f = lambda x : 1/(1+x**2)   # integrand
F = lambda x : np.arctan(x) # antiderivative
a,b = 0,np.inf  # np.Inf, np.infinity, np.Infinity also work
I   = F(b)-F(a)
from scipy.special import roots_laguerre
phi = lambda x : f(x)*np.exp(x)
n   = 185
nodes, weights = roots_laguerre(n)
Q   = np.dot( weights, phi(nodes) )
print(I, Q, I-Q)
```

b) Now repeat the process but try the change of variables $x = e^t - 1$. How small can you make the error? Can you make the error as close to machine precision as you like? *Note:* you are welcome to simplify the integrand by hand (after the change of variables), as this may help with numerical issues.

Solution:

Using $x = e^t - 1$, we have $dx = e^t dt$ and $x = 0 \implies t = 0$ and $x = \infty \implies t = \infty$, so we can write

$$\int_0^\infty \frac{1}{1+x^2}\,dx = \int_0^\infty \frac{1}{1+(e^t-1)^2} e^t\,dt = \int_0^\infty \frac{e^t}{e^{2t}-2e^t+2}\,dt$$

so we will compute $\int_0^\infty \varphi(x) e^{-t}\,dt$ where we define

$$\varphi(x) = e^t \frac{e^t}{e^{2t}-2e^t+2} = \frac{e^{2t}}{e^{2t}-2e^t+2} = \frac{1}{1-2e^{-t}+2e^{-2t}}$$

and now we have the nice fact that $\lim_{x\to\infty} \varphi(x) = 1$ (whereas in part (a), the limit was $+\infty$, since e^x grew faster than our integrand decayed). We may have numerical issues with the accuracy for very large x, but we won't have to take x very large as it turns out, or we could try a more accurate approximation, analogous to how we approximated $e^x - 1$ for $x \approx 0$ earlier in the class.

The code is exactly the same as it was before, except we now define

```
phi = @(x) 1./( 1-2*exp(-x) + 2*exp(-2*x) );
```

and similarly for Python. Note that we needed to simplify the fraction, since any e^t terms would still have led to overflow.

See Fig. 2 (right) for a plot. We can make the error as small as we like (within the confines of machine epsilon $\approx 10^{-15}$), and the best error is for the largest n, though after $n \gtrsim 300$ the error stops going down due to the limits of floating point numbers, so we may as well stop at $n \approx 300$.

Note: both the Matlab and `scipy` Laguerre weights cannot go past about 186, because then the weights have underflow. In Matlab, however, we can go past this n value as you can see in the figure — this is probably because Matlab just ignores these underflows (treats them as zeros), and the error still improves because we're adding more nodes at small x values. In Python, `numpy` doesn't treat these underflow values the same way and probably gives an error, so you cannot continue past about $n = 186$. For both languages, the actual computation of the Laguerre weights is done correctly — it's not the code's fault that the weights are too small to represent in floating point. So, in Python, if you can only go to about $n = 185$, then an error about 10^{-11} is acceptable.

c) Explain the results you observed in parts (a) and (b).

Solution:

Some of this was explained in parts (a) and (b) [and its OK if students did that]. The issue is that our original integrand, the Runge function, doesn't decay exponentially fast, so forcing it to fit in the form of $\int \varphi(x)e^{-x}$ requires that φ grow exponentially fast, and this caused our numerical problems. If we had infinite precision on the computer, then there's nothing wrong with this approach, though the convergence is rather slow.

Our change of variables stretched the x-axis so that with the change of variables, our integrand *did* converge exponentially fast. Now φ doesn't grow at all, and not only are our numerical issues gone, but this "fits" the setting of Gauss-Laguerre better and convergence is very fast. We've approximated integration over a semi-infinite domain $[0, \infty)$ using only about 300 nodes.

Problem 2: High-dimensional integration If z is a continuous random variable with probability density function (pdf) given by $p(z)$, then we define its *mean* or *expected value* as

$$\mathbb{E}[z] = \int_{-\infty}^{\infty} z \cdot p(z)\,dz.$$

That is, the expectation value $\mathbb{E}$ is an integral[2], and expectation values arise a lot: not just for means, but also for variances (since $\mathbb{V}\mathrm{ar}[z] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$) and other quantities. If $z = f(\boldsymbol{x})$ is a function of a *multivariate* random variable $\boldsymbol{x} \in \mathbb{R}^d$, then the expected value is a multidimensional integral

$$\mathbb{E}[f(\boldsymbol{x})] = \int_{\mathbb{R}^d} f(\boldsymbol{x}) \cdot p(\boldsymbol{x}) d\boldsymbol{x}.$$

Because it is common to have multivariate random variables of very large dimension, this means multidimensional integration frequently arises in probability and statistics.

For this problem, use the $d \times d$ symmetric matrix A known as the "Hilbert matrix" defined as $A_{i,j} = 1/(i+j-1)$ for $i, j = 1, \ldots, d$. You can quickly create this in Matlab using `hilb(d)` or in Python using `scipy.linalg.hilbert(d)`. Using this matrix, for $\boldsymbol{x} \in \mathbb{R}^d$, we define

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top A \boldsymbol{x} \tag{1}$$

and choose $\boldsymbol{x} \sim \mathcal{N}(0, I_{d\times d})$ meaning that $\boldsymbol{x}$ is a d-dimensional standard Gaussian random variable, with independent components[3]. That is, if we write the components of $\boldsymbol{x}$ as $\boldsymbol{x} = (x_1, \ldots, x_d)$, then each x_i is a standard normal random variable (mean 0, variance 1) and x_i and x_j are independent if $i \neq j$. The pdf of a standard normal random variable is $p_{\mathrm{N}}(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$. Thus the pdf of $\boldsymbol{x}$ can be written as

$$p(\boldsymbol{x}) = \prod_{i=1}^d p_{\mathrm{N}}(x_i) = \prod_{i=1}^d \frac{1}{\sqrt{2\pi}} e^{-x_i^2/2} = \frac{1}{(2\pi)^{d/2}} \exp\left(\sum_{i=1}^d -x_i^2/2\right).$$

a) Write a code that numerically integrates an arbitrary function in dimension d, for d at least up to 4. Your code can rely on other packages for 1D integration (e.g., in Matlab[4], either `quad` or `integral` — although `quad` is being deprecated in favor of `integral`, I find `quad` is better about being fast if we ask it for low precision — and `scipy.integrate.quad` in Python[5]). However, you may not rely on other packages for 2D, 3D, or other high-dimensional integration (so do not use `integral2` and `integral3` in Matlab, and do not use `scipy.integrate`'s `dblquad`, `tplquad` or `nquad`... except for checking your answers if you want). **Include your code in your homework writeup.** [6]

Solution:

We use `recursiveIntegral.m`, which is included at the end of this document (or see below for an abbreviated version). See the answers to part (b) for how to use this function.

```
function Q = recursiveIntegral( f, d, lwrBnd,uprBnd, quadRule )
if d == 1  % d is the dimension of the integral
    g = f; % base case for recursion
else
    g = @(x) recursiveIntegral( @(varargin)f(x,varargin{:}), d-1, lwrBnd(2:
        end),uprBnd(2:end), quadRule );
end
gg = @(X) arrayfun( g, X );
Q = quadRule( gg, lwrBnd(1), uprBnd(1) );
```

A similar Python version is below:

[2] for discrete random variables, it is a sum not an integral, though you can interpret this sum as a special kind of integral with respect to a different *measure*; e.g., using dirac delta functions.

[3] With this setup, finding $\mathbb{E}[\boldsymbol{x}^\top A\boldsymbol{x}]$ is related to a so-called *Hutchinson* estimator for A.

[4] in Matlab, using `arrayfun` is often convenient, since both `quad` and `integral` will ask to evaluate your function at multiple points simultaneously

[5] in Python, using `np.vectorize` is often convenient, since `quad` will ask to evaluate your function at multiple points simultaneously

[6] You can hardcode f to take 1, 2, 3 or 4 inputs, but if you wanted to make a code that works in any dimensions [not required], you may find it useful to use `varargin` in Matlab or `*args` in Python.

```
#!/usr/bin/env python3
import scipy
import numpy as np
from scipy import integrate
def recursiveIntegral(f, d, lwrBnd, uprBnd, quadRule=scipy.integrate.quad):
    if len(lwrBnd) != d or len(uprBnd) != d:
        raise ValueError('invalid bounds')
    if d == 1:
        g = f    # base case
    else:
        # black magic with *args
        g = lambda x: recursiveIntegral(lambda *args: f(x, *args), d-1,
            lwrBnd[1:], uprBnd[1:], quadRule=quadRule)

    #gg = np.vectorize(g) # slows things down a lot! (close to 15x slower)
    #   Only need to do the np.vectorize if we use quadRule is scipy.
        integrate.quadrature
    #   and the 'vec_func' option is True (which is the default)
    return quadRule(g, lwrBnd[0], uprBnd[0])[0]
    # quad returns tuple (answer, abs_error), only want answer
```

which has doc-string

```
""" Recursively evaluates multivariable integrals
Args:
    f (function): function to evaluate
    d (int): dimension of integral
    lwrBnd (list[float]): lower bounds of integral
    uprBnd (list[float]): upper bounds of integral
    quadRule (function, optional): quadrature function. Defaults to scipy.integrate.quad
"""
```

b) Approximate $\mathbb{E}[f(\boldsymbol{x})]$ using f as defined in Eq. (1) and for $\boldsymbol{x} \sim (0, I_{d\times d})$ for $d = 1, 2, 3, 4$. Although this should be an integral over all of $\mathbb{R}^d$, you may integrate over just the hypercube $[-5, 5]^d$ since $p(\boldsymbol{x})$ is negligible outside this region. To help debug the code, the true integral (over all of $\mathbb{R}^d$) is 1 for $d = 1$, $1.3\overline{3}$ for $d = 2$ and $1.53\overline{3}$ for $d = 3$.

   i. What is the value of the integral for $d = 4$? You should have at least 2 correct digits.

   Solution:

   For $d = 4$, it is $\boxed{\texttt{1.676190476190476}}$. If you're curious, for $d = 5$, it is `1.787301587301587`. In fact, $\mathbb{E}[\boldsymbol{x}^\top A\boldsymbol{x}] = \mathrm{tr}(A)$, the trace of A. [7]

   Matlab code [which was not required to be turned in] is:

```
b = 5; a = -5;d = 4;A = hilb(d);
```

   to setup the problem, then we define f in either of the following two ways (the first way is "fancier" because it works for arbitrary dimension d but requires more programming

[7] If A is a normal matrix, meaning $A^\top A = AA^\top$, then it is unitarily diagonalizable, and we can write down a quick proof that $\mathbb{E}[\boldsymbol{x}^\top A\boldsymbol{x}] = \mathrm{tr}(A)$ [this proof was not required for the homework]. Let $\boldsymbol{v}_i$ and λ_i be the eigenvectors and eigenvalues of A, respectively, so we can write $A = \sum_{i=1}^d \lambda_i \boldsymbol{v}_i\boldsymbol{v}_i^\top$ since $\boldsymbol{v}_i$ are orthonormal, and $\mathbb{E}[\boldsymbol{x}^\top A\boldsymbol{x}] = \sum_{i=1}^d \lambda_i \mathbb{E}[\boldsymbol{x}^\top \boldsymbol{v}_i\boldsymbol{v}_i^\top \boldsymbol{x}]$ by linearity of expectation (i.e., linearity of the integral). But the multivariate Gaussian distribution is invariant under orthogonal changes of variable, and it follows that $\tilde{x} = \boldsymbol{v}_i^\top \boldsymbol{x}$ is distributed like a (univariate) standard normal variable. Then $\mathbb{E}[\boldsymbol{x}^\top \boldsymbol{v}_i\boldsymbol{v}_i^\top \boldsymbol{x}] = \mathbb{E}[\tilde{x}^2] = 1$, hence $\mathbb{E}[\boldsymbol{x}^\top A\boldsymbol{x}] = \sum_{i=1}^d \lambda_i = \mathrm{tr}(A)$. If we had instead chosen $\boldsymbol{x}$ to be a multivariate symmetric Bernoulli distribution, meaning each entry takes ± 1 with equal probability, then it's still true $\mathbb{E}[\boldsymbol{x}^\top A\boldsymbol{x}] = \mathrm{tr}(A)$, and the sample mean of $\boldsymbol{x}^\top A\boldsymbol{x}$ is called the *Hutchinson trace estimator*.

Since the Hilbert matrix is symmetric, hence normal, so I was able to compute the true answer by just calculating $\mathrm{tr}(A)$. Why did I choose the Hilbert matrix for this problem? I wanted a symmetric matrix that is easy to define in any dimension, and such that pattern for its trace, a d increases, is not easy to guess. I excluded diagonal matrices since then the integral becomes separable, in the sense that it is the *product* of d 1D integrals, and this case is too easy.

tricks, namely `cat` and `varargin`, and turns out to run about twice as slow as the "hard-coded" version)

```
% Fancy code; slightly slower (about a factor of 2 slower) ...
f = @(varargin) cat(2,varargin{:})*A*cat(1,varargin{:})*exp( -norm(cat(1,
    varargin{:}))^2/2 )/( (2*pi)^(d/2) );
```

and the alternative, "hard-coded" version

```
switch d
 case 4
   f = @(w,x,y,z) diag([w,x,y,z]*A*[w;x;y;z])'* exp( -[w,x,y,z]*[w;x;y;z]/2 )/( 4*pi^2 );
 case 3
   f = @(x,y,z) diag([x,y,z]*A*[x;y;z])'* exp( -[x,y,z]*[x;y;z]/2 )/( (2*pi)^(d/2) );
 case 2
   f = @(x,y) diag([x,y]*A*[x;y])'* exp( -[x,y]*[x;y]/2 )/( (2*pi)^(d/2) );
 case 1
   f = @(x) x*A*x*exp(-x^2/2)/sqrt(2*pi);
end
```

Either way, we use it like this:

```
tol = 1e-5
quadRule = @(f,a,b) quad(f,a,b,tol);
Q = recursiveIntegral( f, d, a*ones(d,1), b*ones(d,1), quadRule );
```

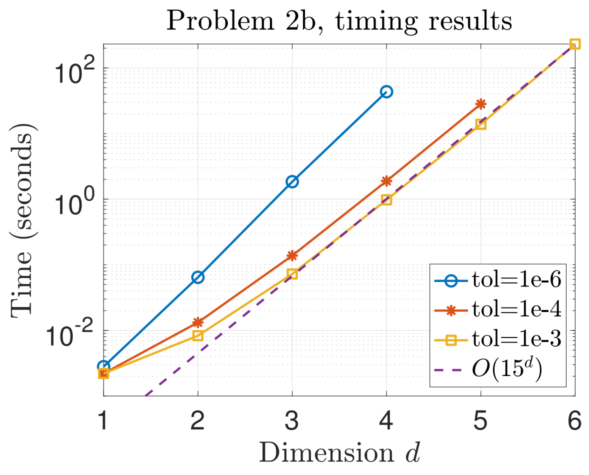
In **Python**, fancy code is

```
#!/usr/bin/env python3
import scipy
import numpy as np
from scipy import integrate
from scipy import linalg
def make_f_function(d):
    const = np.power(2*np.pi, d/2)
    A     = scipy.linalg.hilbert(d)
    def f(*args):
        x = np.atleast_2d(args)
        xt = np.transpose(x) # another way to get transpose
        a = x @ A @ xt
        e = np.exp(-1 * x @ xt / 2) / const
        return a*e
    return f
```

and use it like this:

```
quadRule = lambda *args : scipy.integrate.quad(*args,epsabs=1e-2)
#quadRule = lambda *args : scipy.integrate.quadrature(*args,tol=1e-5,rtol
    =1e-5,vec_func=False)
for d in range(1, 5):
    lwrBnd, uprBnd = [-5]*d, [5]*d
    Q = recursiveIntegral(make_f_function(d), d, lwrBnd, uprBnd, quadRule
         = quadRule)
    print('d : {} ---> {:.12f}'.format(d,Q) )
```

ii. Report the tolerance settings of your 1D integration routine, and then report how long the integral took for each of the dimensions $d = 1, 2, 3, 4$. You can report for larger dimensions if you were able to get the code to work.



```
For d =  1, estimated time is    0.0 seconds
For d =  2, estimated time is    0.0 seconds
For d =  3, estimated time is    0.1 seconds
For d =  4, estimated time is    1.0 seconds
For d =  5, estimated time is   15.2 seconds
For d =  6, estimated time is    3.8 minutes
For d =  7, estimated time is   57.0 minutes
For d =  8, estimated time is   14.2 hours
For d =  9, estimated time is    1.3 weeks
For d = 10, estimated time is   19.1 weeks
For d = 11, estimated time is    5.5 years
For d = 12, estimated time is   82.3 years
For d = 13, estimated time is 1234.3 years
```

Figure 2: Running the multidimensional integration for different dimensions. For the loose tolerance of $1e-3$, errors were $1.6e-3, 1.2e-3, 8.9e-3, 4.5e-2, 9.2e-2, 1.5e-1$ for $d = 1, \ldots, 6$ (so not very good for larger d). Fitting the runtime to a form of $t = c\rho^d$ gives $c \approx 2e-5$ and $\rho \approx 15$; using this, the table on the right estimates the runtime for larger values of d. We can see that it quickly grows out of hand! If any student uses quadrature in dimension $d = 13$ and has good accuracy, then I will be suspicious of their results.

Solution:

Your timing and errors will likely be different than mine, depending on the base 1D quadrature rule you use, though timings should be in a similar ballpark. See Fig. 2 to see how the timing scales [such a figure was not required]. Below is some output.

```
tol = 1e-6
d = 1, estimate is 1.0000, true value is 1.0000, error is 1.56e-05, took 0.003 seconds
d = 2, estimate is 1.3333, true value is 1.3333, error is 2.40e-05, took 0.064 seconds
d = 3, estimate is 1.5333, true value is 1.5333, error is 2.42e-05, took 1.846 seconds
d = 4, estimate is 1.6762, true value is 1.6762, error is 3.19e-05, took 43.417 seconds

tol = 1e-4
d = 1, estimate is 1.0000, true value is 1.0000, error is 2.74e-05, took 0.002 seconds
d = 2, estimate is 1.3329, true value is 1.3333, error is 4.03e-04, took 0.015 seconds
d = 3, estimate is 1.5328, true value is 1.5333, error is 5.61e-04, took 0.145 seconds
d = 4, estimate is 1.6717, true value is 1.6762, error is 4.52e-03, took 1.918 seconds
d = 5, estimate is 1.7605, true value is 1.7873, error is 2.68e-02, took 27.811 seconds

tol = 1e-3
d = 1, estimate is 0.9984, true value is 1.0000, error is 1.59e-03, took 0.002 seconds
d = 2, estimate is 1.3321, true value is 1.3333, error is 1.26e-03, took 0.008 seconds
d = 3, estimate is 1.5245, true value is 1.5333, error is 8.86e-03, took 0.072 seconds
d = 4, estimate is 1.6306, true value is 1.6762, error is 4.56e-02, took 0.976 seconds
d = 5, estimate is 1.6888, true value is 1.7873, error is 9.85e-02, took 13.858 seconds
d = 6, estimate is 1.7262, true value is 1.8782, error is 1.52e-01, took 231.810 seconds
```

c) Sometimes we use quadrature methods to evaluate expectations, but we can also use expectations to evaluate integrals for us: this is called Monte Carlo estimation. Repeat your estimate of $\mathbb{E}[f(\boldsymbol{x})]$ for $d = 1, 2, 3, 4$ by computing the *sample mean* of $f(\boldsymbol{x})$ over n different independent realizations of $\boldsymbol{x}$. You can draw a realization of $\boldsymbol{x}$ using `randn(d,1)` in Matlab, or `numpy.random.randn(d)` in Python[8]. Run $n = 10^6$ realizations to form your Monte Carlo estimator, and report both your *runtime* (in seconds) and *error*, for $d = 4$,

[8]though the recommended way is `from numpy.random import default_rng` then `rng = default_rng()` then `rng.standard_normal(d)`.

$d = 12$ and $d = 100$. For $d = 4$, you can use the answer you found via quadrature as the "true answer" for calculating the error; for $d = 12$, the true value is `2.224352838648`, and for $d = 100$ it is `3.284342189302`.

Solution:

The time and accuracy will both vary; the time depends on your implementation (see below for two possible implementations), and the accuracy is a random variable so this will vary. You can see how the accuracy is variable by looking at the plots [students were not required to turn in plots] in Fig. 3.

An example of a reasonable output is below:

```
d=4, basic Monte Carlo:   0.472 seconds for n=1e+06 iterates, error is 2.44e-03
d=4, faster Monte Carlo:  0.077 seconds for n=1e+06 iterates, error is 5.10e-03
d=12, basic Monte Carlo:  0.567 seconds for n=1e+06 iterates, error is 3.02e-03
d=12, faster Monte Carlo: 0.151 seconds for n=1e+06 iterates, error is 2.21e-04
d=100, basic Monte Carlo: 2.232 seconds for n=1e+06 iterates, error is 4.23e-03
d=100, faster Monte Carlo:1.407 seconds for n=1e+06 iterates, error is 8.16e-04
```

A basic Monte Carlo implementation is:

```
S = 0;nRep = 1e6;Qlist = zeros(nRep,1);A = hilb(n)for n = 1:nRepX =
    randn(d,1);Q = X'*A*X;S = S + Q;Qlist(n) = S/n;end
```

and a much faster implementation (but requires more memory, and harder to follow perhaps) is

```
nRep    = 1e6;
A               = hilb(n)
X       = randn(d,nRep);
AX      = A*X;
z       = sum( X.*AX, 1);
Qlist   = cumsum(z)./(1:nRep);
```

Similar code in Python would look like

```
#!/usr/bin/env python3
import scipy.linalg
A = scipy.linalg.hilbert(4)
import numpy as np
n = int(1e6)
x = np.random.randn(4,n)
z = sum(x * ( A @ x))
Q = np.mean(z)
```

and if you wanted to plot it in Python,

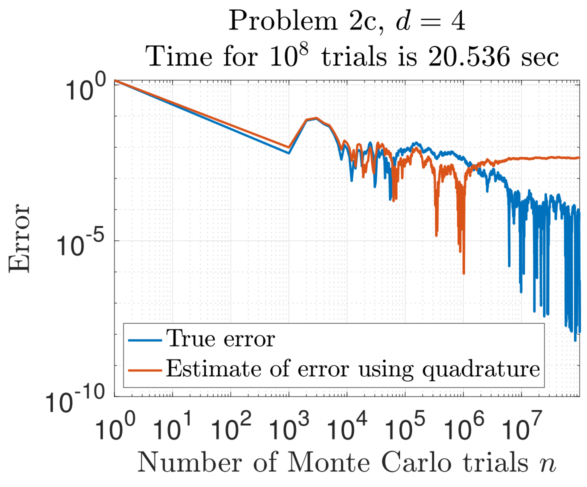
```
Qlist = np.cumsum(z)/np.arange(1,n+1)
import matplotlib.pyplot as plt
plt.loglog( np.abs(Qlist - np.trace(A)) )
```

d) Make some informed comments about the difference between the quadrature and Monte Carlo methods

Solution:

There is wide latitude in what is acceptable, but generally we're looking for something mentioning at least one of the following observations:

i. Quadrature rules get extremely slow in high dimensions



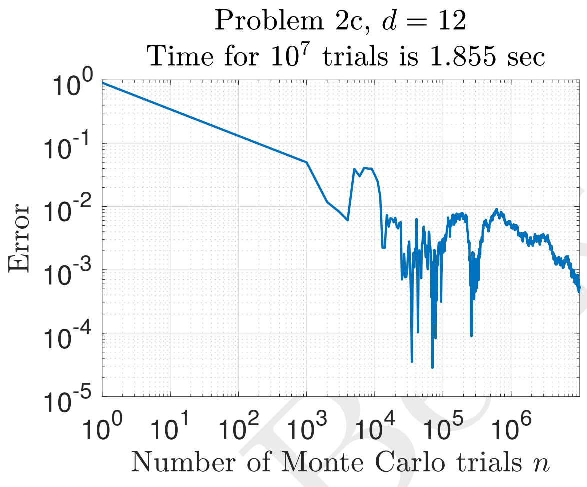


Figure 3: Monte Carlo error, for $d = 4$ (left) and $d = 12$ (right), as a function of number of trials n. Each trial is very fast, but it takes many trials to reach acceptable error levels. Monte Carlo integration is great for low-accuracy solutions in high dimensions, but numerical quadrature rules are much faster and accurate in 1D.

ii. Monte Carlo works well even in high dimension

iii. Monte Carlo is not that high accuracy though: it takes many iterations to get low accuracy

iv. In 1D, quadrature rules get very high accuracy much easier than Monte Carlo

Optional problem (Not graded) Estimate

$$I = \lim_{\epsilon \to 0} \int_\epsilon^1 x^{-1} \cos(x^{-1} \ln(x))\, dx$$

as accurately as possible using a numerical scheme, and describe your method. In previous years, students received 1 point of credit for every accurate digit of their solution.

Background on Gauss-Laguerre quadrature

The Laguerre polynomials $\{L_n\}_{n=1}^\infty$ are defined, up to a scaling constant, such that L_n is a degree n polynomial and that

$$\int_0^\infty L_n(x) L_m(x) e^{-x}\, dx = 0 \quad \forall n, m \in \mathbb{N}, n \neq m.$$

In other words, the polynomials are *orthogonal* on the domain $[0, \infty)$ with the weight function $w(x) = e^{-x}$ (we cannot have $w(x) = 1$ because the integral over $[0, \infty)$ of any polynomials, other than the 0 polynomial, is infinite). To resolve the scaling ambiguity, some scaling convention is used, such as requiring them to be monic (i.e., leading coefficient is 1), or that $\int_0^\infty L_n(x) e^{-x}\, dx = 1$; the scaling does not concern us since we'll only care about their roots. These polynomials arise in many areas of math, and are the solutions to the 2nd order linear ODE $xy'' + (1 - x)y' + ny = 0$, and arise as part of the solution to the Schrödinger equation of the one-electron atom in quantum mechanics.

We'll use the fact that L_n has n simple real roots in $(0, \infty)$. We use these as the nodes x_i. Our goal is to approximate an integral of the form

$$I(f) = \int_0^\infty f(x) e^{-x}\, dx \tag{2}$$

which we will do by interpolating f with a degree $n-1$ polynomial on n nodes; this polynomial p is unique, and is the Lagrange interpolating polynomial we've seen before. Then we approximate I by $I_n(f) = \int_0^\infty p(x) e^{-x}$;

this integral I_n is tractable since we can write $p(x)$ as a weighted sum of monomials, and for any monomial x^k, we can determine $\int_0^\infty x^k e^{-x}\,dx$ by doing integration by parts k times (though there are shortcuts/formulas; we don't actually do integration by parts in practice when numerically evaluating this). It follows that if f itself is a degree $n-1$ or less polynomial, then f is its own interpolatory polynomial of degree $n-1$ or less, and so $I(f) = I_n(f)$. Now suppose f is a polynomial of degree between n and $2n-1$. Divide f by the Laguerre polynomial L_n, so we can write $f(x) = Q(x)L_n(x) + R(x)$ using polynomial long division, where R has degree less than L_n (so strictly less than n) and Q has degree between 0 and $n-1$. Then

$$\begin{aligned}
\int_0^\infty f(x)e^{-x}\,dx &= \underbrace{\int_0^\infty Q(x)L_n(x)e^{-x}\,dx}_{=0 \text{ by orthogonality}} + \int_0^\infty R(x)e^{-x}\,dx \\
&= 0 + I(R) \\
&= I_n(R) \text{ since } R \text{ has degree } n-1 \text{ or less} \\
&= I_n(f) \text{ since } f(x_i) = Q(x_i)\cdot \underbrace{L_n(x_i)}_{=0} + R(x_i) = R(x_i)
\end{aligned}$$

Matlab doesn't have a builtin routine to compute the weights and zeros for Gauss-Laguerre quadrature, but the well-respected `Chebfun` package does: see `lagpts.m`, which relies on `gammaratio.m`. You can download these directly from Matlab using the following commands:

```
    websave('lagpts.m','https://github.com/chebfun/chebfun/raw/master/lagpts.m')
    websave('gammaratio.m','https://github.com/chebfun/chebfun/raw/master/gammaratio.m');
```

Python's `scipy` package has routines for the weights and zeros. Use `scipy.special.roots_laguerre` or `numpy.polynomial.laguerre.laggauss` (which looks like it may not be as tested as the `scipy` version).

Problem 2 complete code

Here is the code for Problem 2:

```
function Q = recursiveIntegral( f, d, lwrBnd,uprBnd, quadRule )
% Q = recursiveIntegral( f, d, lwrBnd,uprBnd, quadRule )
%   computes the integral of f in R^d using quadRule as the base 1D
%   integration routine. quadRule should take in (f,a,b) as inputs.
%   lwrBnd and uprBnd are vectors of length d
if nargin < 5 || isempty(quadRule)
    quadRule = @integral;
end
if numel(lwrBnd) ~= d || numel(uprBnd) ~= d
    error('Lower and upper bounds should have be length d');
end

% == Fancy version using varargin:
if d == 1
    g = f;
else
    g = @(x) recursiveIntegral( @(varargin)f(x,varargin{:}), d-1, lwrBnd(2:end),uprBnd(2:end
        ), quadRule );
end
% == Hard-coded version (also acceptable, but limits to d <= 4)
% switch d
%     case 4
%         g = @(w) recursiveIntegral( @(x,y,z)f(w,x,y,z), d-1, lwrBnd(2:end),uprBnd(2:end),
    quadRule );
%     case 3
%         g = @(x) recursiveIntegral( @(y,z)f(x,y,z), d−1, lwrBnd(2:end),uprBnd(2:end),
    quadRule );
%     case 2
%         g = @(y) recursiveIntegral( @(z)f(y,z), d−1, lwrBnd(2:end),uprBnd(2:end), quadRule
     );
%     case 1
%         g = f;
% end


gg = @(X) arrayfun( g, X );
Q = quadRule( gg, lwrBnd(1), uprBnd(1) );
```